Assignment 4 (CS 432)
Lightweight DBMS with B+ Tree Index

8th April, 2025

Objective

Design and implement a lightweight database management system (DBMS) in Python that supports basic operations (insert, update, delete, select, aggregation, range queries) on tables stored with a B+ Tree index.

1 Introduction

Efficient data storage and retrieval are fundamental challenges in computer science, particularly in database systems and file indexing. The B+ Tree is a self-balancing tree structure that enhances performance in disk-based and memory-based data management. It optimizes search, insertion, and deletion operations, making it widely used in database indexing, file systems, and key-value stores.

This assignment focuses on implementing a B+ Tree with essential operations, including insertion, deletion, search, and range queries. Furthermore, a performance analysis is conducted by comparing the B+ Tree with a brute-force approach. This will provide insights into how structured indexing improves efficiency.

For a fundamental understanding of the B+ tree, refer to the lecture 22 slides Lecture 22 slides.

For More Understanding of the B+ tree and its implementation, this YouTube playlist might be helpful: B+ Trees.

2 Implementation Details

2.1 B+ Tree Features

Implement a B+ Tree with the following functionalities:

- **Insertion:** Keys are inserted while ensuring automatic node splitting.
- **Deletion:** Keys are removed with proper merging and redistribution.
- **Exact Search:** Ability to find whether a key exists in the tree.
- **Range Queries:** Retrieve all keys within a given range.
- **Value Storage:** Associate values (records for the table) with keys in the tree.

2.2 Performance Analysis

Compare the B+ Tree with BruteForceDB approach by measuring:

- **Insertion Time:** How long it takes to insert keys in both structures.
- **Search Time:** Compare the time taken for exact match searches.
- **Deletion Time:** Compare the time taken to delete the records.
- **Range Query Time:** Measure the efficiency of retrieving keys in a range.
- **Random Performance:** Performance of Random task (Insertion, Search, Deletion).
- **Memory Usage:** Track how much memory is used by each structure.
- **Automated Benchmarking:** Conduct multiple tests and collect results.

2.3 Visualization

Use Graphviz to visualize the tree structure:

- **Tree Structure:** Show the hierarchy of internal and leaf nodes.
- **Node Relationships:** Display parent-child relationships.
- **Leaf Node Linkage:** Highlight linked list connections in leaves.

3 Implementation Tasks

File Structure

```
db_management_system/
├── database/
│   ├── __init__.py
│   ├── db_manager.py ........................................ Database manager
│   ├── table.py ........................................ Table implementation
│   ├── bplustree.py .................................... B+ Tree implementation
│   └── bruteforce.py ............................................ BruteForceDB
├── report.ipynb ................................... Report and Visualizations
└── requirements.txt ................................... Project dependencies
```

Task 1: Implement the B+ Tree

- Define `BPlusTreeNode` and `BPlusTree` classes.
- Implement insertion, deletion, search, and range queries.
- Ensure automatic splitting and merging of nodes.

Task 2: Implement Performance Analysis

- Create the `PerformanceAnalyzer` class.
- Implement methods to measure time complexity and memory usage.
- Compare the B+ Tree with brute-force approach (`BruteForceDB`).

Task 3: Implement Visualization

- Use Graphviz (`graphviz.Digraph`) to visualize the tree.
- Implement node connections and leaf node linkages.

Task 4: Conduct Performance Testing

- Generate different set sizes of random keys (e.g., using `range(100, 100000, 1000)`).
- Insert random key sets of different sizes into both data structures.
- Again generate random key sets, perform search operations, and measure execution time.
- Do the same for range queries and delete operations; compare efficiency.
- Visualize the results using Matplotlib plots.

Task 5: Implement Database Class

- Create a class named `Database`.
- This class should have the functionality to create and manage tables (e.g., creating a new table, deleting a table, listing tables).

Task 6: Make database Persistent

Store any 2 tables from the database implemented in module 1. You need not worry about the referential integrity between them.
Make the database persistent so that even when the program ends, the database is stored securely in the disk. Also include functionality to retrieve the stored database.

Task 7: Write a Report

Test the database created in a jupyter notebook and document the following in it.

- **Introduction:** Explain the problem addressed and the proposed solution (B+ Tree DBMS).
- **Implementation:** Describe the implementation details of the B+ Tree operations.
- **Performance Analysis:** Present the benchmarking results using tables and graphs. Discuss the findings.
- **Visualization:** Include results of the generated tree visualizations of your database tables.
- **Conclusion:** Summarize the project findings, challenges, and potential future improvements.

4 Evaluation Criteria

Criteria	**Marks**
Implementation B+ Tree (insertion, deletion, search)	20
Implementation of range queries and value storage	20
Automated benchmarking and performance analysis (time/memory comparison)	20
Visualization using Graphviz	10
Persistence	10
Store 2 tables from your module 1 database using current implementation	10
Report and discussions	10
Create a UI to make Operations and Visualisation on current DB implementation	**BONUS**
Total	**100** + BONUS(**30**)

NOTE: To get bonus points, you need to develop a UI where –

- We can perform all the mentioned actions on DB as per the given task.
- Select and check data for created tables.

5 Submission Guidelines

Submit a private Github repository link containing:

- A `database` folder with logic to implement the required functionalities .
- A report in the format of ipynb file (jupyter notebook).
- **Submission Deadline 22nd April Midnight**

A bplustree.py

This appendix provides the boiler plate code for the `bplustree.py`. There may be other functions that are required, implement them accordingly.

```python
def search(self, key):
    # Search for a key in the B+ tree. Return associated value if found, else None
    .
    # Traverse from root to appropriate leaf node.
    pass

def insert(self, key, value):
    """
    Insert key-value pair into the B+ tree.
    Handle root splitting if necessary.
    Maintain sorted order and balance properties.
    """
    pass

def _insert_non_full(self, node, key, value):
    # Recursive helper to insert into a non-full node.
    # Split child nodes if they become full during insertion.
    pass

def _split_child(self, parent, index):
    """
    Split parent's child at given index.
    For leaves: preserve linked list structure and copy middle key to parent.
    For internal nodes: promote middle key and split children.
    """
    pass

def delete(self, key):
    """
    Delete key from the B+ tree.
    Handle underflow by borrowing from siblings or merging nodes.
    Update root if it becomes empty.
    Return True if deletion succeeded, False otherwise.
    """
    pass

def _delete(self, node, key):
    # Recursive helper for deletion. Handle leaf and internal nodes.
    # Ensure all nodes maintain minimum keys after deletion.
    pass

def _fill_child(self, node, index):
    # Ensure child at given index has enough keys by borrowing from siblings or
    merging.
    pass

def _borrow_from_prev(self, node, index):
    # Borrow a key from the left sibling to prevent underflow.
    pass

def _borrow_from_next(self, node, index):
    # Borrow a key from the right sibling to prevent underflow.
    pass

```

```python
def _merge(self, node, index):
    # Merge child at index with its right sibling. Update parent keys.
    pass

def update(self, key, new_value):
    # Update value associated with an existing key. Return True if successful.
    pass

def range_query(self, start_key, end_key):
    """
    Return all key-value pairs where start_key <= key <= end_key.
    Traverse leaf nodes using next pointers for efficient range scans.
    """
    pass

def get_all(self):
    # Return all key-value pairs in the tree using in-order traversal.
    pass

def visualize_tree(self):
    # Generate Graphviz representation of the B+ tree structure.
    pass

def _add_nodes(self, dot, node):
    # Recursively add nodes to Graphviz object (for visualization).
    pass

def _add_edges(self, dot, node):
    # Add edges between nodes and dashed lines for leaf connections (for
    visualization).
    pass
```

Listing 1: Required B+ Tree methods

B BruteForceDB

This appendix provides the Python code for the BruteForceDB class, which serves as a baseline for performance comparison against the B+ Tree. It uses a simple list to store keys and performs operations through linear iteration.

```python
class BruteForceDB:
    def __init__(self):
        self.data = []

    def insert(self, key):
        self.data.append(key)

    def search(self, key):
        return key in self.data

    def delete(self, key):
        if key in self.data:
            self.data.remove(key)

    def range_query(self, start, end):
        return [k for k in self.data if start <= k <= end]
```

Listing 2: BruteForceDB Class (bruteforce.py)